Template

Motivation

Many C++ programs use common data structures like stacks, queues and lists[1].

A program may require a queue of *customers* and a queue of *messages*. One could easily implement a queue of *customers*, then take the existing code and implement a queue of *messages*.

Later you might need a queue of *orders*. So just take the queue of *messages* and convert that to a queue of *orders* (through copy, paste, find, and replace).

What if you need to make some changes to all of the queue implementations (*customer*s, *message*s, and *order*s)? Not so easy now with all of these copies.

Re-inventing source code is not an intelligent approach in an object oriented environment which encourages re-usability.

It seems to make more sense to implement a queue that can contain any arbitrary type rather than duplicating code.

This is when type parameterization, commonly referred to as **template**, comes into play.

Intro to Templates

C++ templates allow one to implement a generic Queue<T> template that has a type parameter T. T is generic and can be placed with actual types.

For example, with Queue<Customers>, C++ will generate the class Queue<Customers>. Changing the implementation of the Queue becomes relatively simple. Once the changes are implemented in the template Queue<T>, they are reflected in the classes Queue<Customers>, Queue<Messages>, and Queue<Orders> when recompiled.

Templates provide a way to re-use source code, different from inheritance and object composition.

[1] This lecture is based on this very nice tutorial http://users.cis.fiu.edu/~weiss/Deltoid/vcstl/templates

C++ provides two kinds of templates: class templates and function templates. The Standard Template Library containers have been implemented as class templates, and STL generic algorithms have been implemented as function templates.

Class Templates

A class template definition is similar to a regular class definition, except it is prefixed by the keyword template. For example:

```
template <class T>
class Stack{
public:
    Stack(int = 10);
    ~Stack(){ delete [] stackPtr; }
    int push(const T&);
    int pop(T&);
    int isEmpty() const{ return top == -1; }
    int isFull() const{ return top == size - 1; }
private:
    int size;  // number of elements on Stack.
    int top;
    T* stackPtr;
};
```

T is a type parameter and it can be any type. For example, Stack<Token>, where Token is a user defined class. T does not have to be a class type as implied by the keyword class. For example, Stack<int> and Stack<Message*> are valid instantiations, even though int and Message* are not "classes".

Class template member functions

The declarations and definitions of the class template member functions need to be in the same header file.

Implementing template member functions is somewhat different compared to the regular class member functions. Consider the following wrong example.

B.h (Wrong)

```
template <class t>
class b{
public:
    b() ;
    ~b() ;
} ;
```

B.cpp (Wrong)

```
#include "B.H"
template <class t>
b<t>::b(){}
template <class t>
b<t>::~b(){}
```

main.cpp

```
#include "B.H"
void main(){
    b<int> bi ;
    b <float> bf ;
}
```

Previously, we have discussed that each .cpp will be compiled independently first, then linked.

When compiling B.cpp, the compiler has both the declarations and the definitions available. At this point the compiler does not need to generate any definitions for template classes, since there are no instantiations: no one is using the template. Things are still fine.

When the compiler compiles main.cpp, there are two instantiations: template class B<int> and B<float>. At this point the compiler needs the definition to generate those instantiations. But there are only declarations in main.cpp but no definitions! The compiler cannot generate any definitions for these instantiations.

While implementing class template member functions, the definitions are prefixed by the keyword template. Here is the complete implementation of class template Stack:

stack.h

```
#pragma once
template <class T>
class Stack{
public:
    Stack(int = 10);
    ~Stack(){ delete [] stackPtr ; }
    int push(const T&);
    int pop(T&);  // pop an element off the stack
    int isEmpty() const { return top == -1; }
    int isFull() const { return top == size - 1; }
private:
    int size;  // Number of elements on Stack
```

```
    int top;
    T* stackPtr;
} ;

//constructor with the default size 10
template <class T>
Stack<T>::Stack(int s){
    size = s > 0 && s < 1000 ? s : 10 ;
    top = -1 ;  // initialize stack
    stackPtr = new T[size] ;
}

 // push an element onto the Stack
template <class T>
int Stack<T>::push(const T& item){
    if (!isFull()){
        stackPtr[++top] = item ;
        return 1 ;  // push successful
    }
    return 0 ;  // push unsuccessful
}

// pop an element off the Stack
template <class T>
int Stack<T>::pop(T& popValue) {
    if (!isEmpty()){
        popValue = stackPtr[top--] ;
        return 1 ;  // pop successful
    }
    return 0 ;  // pop unsuccessful
}
```

Using a class template

Create the required classes by providing the actual type for the type parameters. This process is commonly known as "Instantiating a class".

```
#include <iostream>
#include "stack.h"

using namespace std ;

void main(){
    typedef Stack<float> FloatStack;
    typedef Stack<int> IntStack;

    FloatStack fs(5) ;
    float f = 1.1 ;
    cout << "Pushing elements onto fs" << endl ;
    while (fs.push(f)){
        cout << f << ' ' ;
        f += 1.1 ;
    }
```

```
    cout << endl << "Stack Full." << endl
    << endl << "Popping elements from fs" << endl ;
    while (fs.pop(f))
       cout << f << ' ' ;
    cout << endl << "Stack Empty" << endl ;
    cout << endl ;

    IntStack is ;
    int i = 1.1 ;
    cout << "Pushing elements onto is" << endl ;
    while (is.push(i)){
       cout << i << ' ' ;
       i += 1 ;
    }
    cout << endl << "Stack Full" << endl
    << endl << "Popping elements from is" << endl ;
    while (is.pop(i))
          cout << i << ' ' ;
    cout << endl << "Stack Empty" << endl ;
}
```

In the above example we defined a class template Stack. In main we instantiated a Stack of float (`FloatStack`) and a Stack of int (`IntStack`). Once the template classes are instantiated you can instantiate objects of that type (for example, fs and is.)

A good programming practice is using `typedef` while instantiating template classes. Then throughout the program, one can use the `typedef` name.

With `typedef`:

```
    typedef vector<int, allocator<int> > INTVECTOR;
    INTVECTOR vi1;
    INTVECTOR vi2;
    INTVECTOR vi2;
```

Without `typedef`:

```
    vector<int, allocator<int> >  vi1;
    vector<int, allocator<int> >  vi2;
    vector<int, allocator<int> >  vi2;
```

There are two advantages using `typedef`:

1. `typedef`'s are very useful when "templates of templates" come into usage. For example, when instantiating an STL vector of int's, you could use:

```
    typedef vector<int, allocator<int> > INTVECTOR;
    INTVECTOR vi1;
```

2. If the template definition changes, simply change the typedef definition.

```
    typedef vector<int> INTVECTOR;
    INTVECTOR vi1;
```

Function Templates

To perform identical operations for each type of data compactly and conveniently, use function templates.

```
#include <iostream>

using namespace std ;

//max returns the maximum of the two elements
template <class T>
T max(T a, T b){
    return a > b ? a : b ;
}
```

Based on the argument types provided in calls to the function, the compiler automatically instantiates separate object code functions to handle each type of call appropriately.

```
void main(){

    cout << "max(10, 15) = " << max(10, 15) << endl ;
    cout << "max('k', 's') = " << max('k', 's') << endl ;
    cout << "max(10.1, 15.2) = " << max(10.1, 15.2) << endl ;
}
```

When the compiler sees an instantiation of the function template, for example the call max(10, 15) in function main, the compiler generates a function max(int, int). Similarly the compiler generates definitions for max(char, char) and max(float, float) in this case.

We can have multiple template type parameters too[2]:

```
#include <iostream>

template <class T, class U>
auto max(T x, U y){ // x resolves to T, and y resolves to U
    return (x > y) ? x : y;
}

int main(){
    std::cout << max(2, 3.5) << '\n';
```

[2] https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/

```
    return 0;
}
```

Why use auto as the return type? What should be return when we input an int and a double?

Template Instantiation

The creation of a new definition of a function, class, or member of a class from a template declaration and one or more template arguments is called template instantiation[3].

Template instantiation has two forms: **explicit instantiation** and **implicit instantiation**.

You can explicitly tell the compiler when it should generate a definition from a template. This is called explicit instantiation.

```
template <class T>
class Z{
public:
    Z() {} ;
    ~Z() {} ;
    void f(){} ;
    void g(){} ;
} ;

int main(){
    template class Z<int>; //explicit instantiation
    template class Z<float>; //explicit instantiation

    …
    return 0 ;
}
```

Unless a template specialization has been explicitly instantiated or explicitly specialized, the compiler will generate a specialization for the template only when it needs the definition. This is called implicit instantiation.

```
template <class T>
class Z{
public:
    Z() {} ;
    ~Z() {} ;
    void f(){} ;
    void g(){} ;
```

[3] Template instantiation https://www.ibm.com/docs/en/i/7.3?topic=only-template-instantiation-c

```
} ;

int main(){
    Z<int> zi ;   //implicit instantiation generates class Z<int>
    zi.f() ;    //and generates function Z<int>::f()
    Z<float> zf ;//implicit instantiation generates class Z<float>
    zf.g() ;     //and generates function Z<float>::g()
    return 0 ;
}
```

Class Template Specialization

The definition created from a template instantiation to handle a specific set of template arguments is called a specialization.

It is possible to override the template-generated code by providing special definitions for specific types. This is called **template specialization**. The following example defines a template class specialization for template class stream to specialize `T` to `char`.

```
#include <iostream>
using namespace std ;

template <class T>
class stream{
    public:
        void f() { cout << "stream<T>::f()"<< endl ;}
} ;

template <>
class stream<char>{
    public:
        void f() { cout << "stream<char>::f()"<< endl ;}
} ;

int main(){
    stream<int> si ;
    stream<char> sc ;

    si.f() ;
    sc.f() ;
}
```

Template parameters

```
template <class T>
class Stack{
} ;
```

Here `T` is a template parameter, also referred to as **type-parameter**. In addition, templates could have **non-type parameters**.

```
#include <iostream>

template <typename T, int size> // size is a non-type parameter
class StaticArray{
private:
    T m_array[size] {};

public:
    T* getArray();

    T& operator[](int index){
        return m_array[index];
    }
};

template <typename T, int size>
T* StaticArray<T, size>::getArray(){
    return m_array;
}

int main(){
    // declare an integer array with room for 12 integers
    StaticArray<int, 12> intArray;

    // declare a double buffer with room for 4 doubles
    StaticArray<double, 4> doubleArray;

    //...
}
```

C++ allows you to specify a default template parameter, and the definition looks like:

```
template <class T = float, int elements = 100> Stack { ....} ;
```

Then a declaration such as

```
Stack<> mostRecentSalesFigures ;
```

would instantiate (at compile time) a 100 element Stack template class named mostRecentSalesFigures of float values.

Note since templates are instantiated at compile time (not run time), the template non-type parameter has to be of constant expression. In other words, it cannot be some value that will change during run time.

Static Members and Variables

Each template class or function generated from a template has its own copies of any static variables or members.

```
#include <iostream>

using namespace std;

template <class T>
class X{
public:
    static T s ;
};

template<> int X<int>::s = 3;
template<> char X<char>::s = 'a';

int main(){
    X<int> xi, xj; // xi and xj share X<int>::s
    X<char> xc ; // xc has its own static var X<char>::s

    cout << X<int>::s << endl;
    cout << X<char>::s << endl;
}
```

Here X<int> has a static data member s of type int, and X<char> has a static data member s of type char.

Templates and Friends

You can also have friend in templates. In the following case,

```
template <class T>
class X{
    friend void f1();
    …
} ;
```

f1() is friends of all instantiations of template X. For example, f1() is a friend of X<int>, X<A>, and X<Y>.

However, for the following case

```
template <class T>
class X{
    friend void f2(X<T>&) ;
    …
} ;
```

The friend function is not friends to all instantiations. For a particular type T for example, float, makes f2(X<float>&) a friend of class X<float> only. The f2(x<float>&) cannot be a friend of class X<A>.